(Preprint) AAS 23-XXX

A NEW ARCHITECTURE FOR PARALLELIZATION OF COMPLEX SPACECRAFT TRAJECTORY OPTIMIZATION SCANS

Quentin Moore [*], Brian Killeen [†], Jacob Williams [‡]

This paper describes CopScanner, a new component of the Copernicus ecosystem for spacecraft trajectory design and optimization. CopScanner is a Python library being developed at the NASA JSC which enables easy parallelization of Copernicus scans. CopScanner is currently being developed and implemented for production of Copernicus trajectory scans for upcoming Artemis Missions (Artemis II and beyond). On the backend, CopScanner utilizes Dask, an open-source Python library for parallel computing which enables parallelization over both multi-core local machines and large-scale distributed computing clusters. CopScanner abstracts the trajectory scanning process into a DAG which is constructed using a chain of individual subscans. Each node in the DAG executes a python module, called the callable, for which there are built-in defaults, or users may specify their own. Support for custom callables makes CopScanner a versatile trajectory optimization software. All output files and associated metadata from a CopScanner scan are compressed and stored in a two-file output, collectively called the FileStore, consisting of a SQLite database and a compressed JSON MessagePack file, for which CopScanner provides a Python class for interaction.

INTRODUCTION

A common facet of modern trajectory design is the generation of many trajectories in a short amount of time. Advances in computational power and the rise of high performance computing have enabled engineers to generate thousands of high fidelity trajectories within hours. This includes trajectories generated with an optimizer such as Copernicus.[1] The process of solving many trajectories in parallel, with possible dependencies on each other, is refered to as a "trajectory scan". In this paper, it refers to a number of trajectories that are solved in parallel and can be traced to a common, initial trajectory. Trajectory scans often have a series of parameters that are being varied in the scanning process, such as launch epoch, vehicle mass, or landing site.

Many spaceflight programs and missions have a need to execute large scale trajectory scans, including the Artemis missions,[2–4] CAPSTONE,[5] and Gateway's Power and Propulsion Element (PPE).[6] For the Artemis I mission, the trajectory scan capability was initially heavily focused on solving trajectories that represented the nominal mission. There was a several step process in order to produce trajectories that modeled a single launch opportunity:[4]

1. **A Day Scan**: For Earth-Moon transfers (e.g., for the Artemis missions) there will be an optimal Earth Moon transfer once about every 24.84 hours. If an initial guess is available for

[*]Odyssey Space Research, Houston, TX 77058
[†]EG/Aeroscience and Flight Mechanics, NASA Johnson Space Center, Houston, TX 77058
[‡]EG/Aeroscience and Flight Mechanics, NASA Johnson Space Center, Houston, TX 77058

each case, they can all run in parallel (See HLS[7]). Otherwise, a continuation method may be used, where each case uses the converged solution from the previous case as an initial guess.
2. **A Selection Scan**: Artemis I could fly a series of possible Orion missions after Trans-Lunar Injection (TLI), however a single mission had to be selected for generation of launch windows.
3. **A Launch Window Scan**: a daily launch window is constructed from the optimal point. For the Artemis I mission, a continuation method was used from the optimal point, marching forward and backward in time from the optimal in order to construct all the solutions for the full window from open to close.

After the trajectory scan logic to generate nominal trajectories for Artemis I was developed, there was also a need to execute trajectory scans for various off-nominal scenarios.[8] Each off-nominal trajectory scenario would be solved once for each nominal trajectory that was generated by the trajectory scan:

- *Aborts and Early Returns*: Trajectories to initiate a return to Earth Entry Interface (EI). These were solved for many different burn ignition times, burn engines, EI days, etc.
- *Missed Burn Recoveries*: Trajectories built to recover the spacecraft after a major burn was missed, or partially completed.
- *Alternate Missions*: In the event that TLI did execute as intended, a variety of alternate Orion missions were generated to satisfy the top mission priorities.

The broad variety of off-nominal trajectories, all with different input parameters that needed to be varied in the scan, made building a targeted software to perform scanning logic difficult. To complicate the matter further, some of these off-nominal trajectories required an initial guess of a previous off-nominal solution (e.g., an abort solution at a particular Time of Ignition (TIG) required the solved abort at a TIG 12 hours previous). Other off-nominal trajectories simply required the nominal trajectory solution and were massively parallel. The varying dependencies between cases made writing specific logic for a particular problem difficult. Therefore, a generalized architecture for executing any Copernicus scan given input scan parameters and a Python method to solve the trajectory was built. The generalized architecture was successful in generating millions of off-nominal trajectories for Artemis I, and was developed into a tool to be incorporated into the Copernicus ecosystem. The release of Copernicus 5.0[1] in 2020 refactored the software into a shared library that can be called by Python. This allowed the growth of CopScanner and an opportunity for inclusion in the Copernicus ecosystem.

The original prototype software, which was derived from work done on Artemis, grew organically, but it was rooted in basic graph theory techniques which allowed for the expansion of the trajectory scan generalization that can be accomplished. Graph theory is a well-known technique for certain astrodynamics problems;[9] however, the approach detailed in this paper replaces complex, ad-hoc parallelization schemes with a novel, generalized method of solving trajectory scans. This paper will further investigate the parallelization package that has been developed for executing trajectory scans.

DASK

The core building block of CopScanner is Dask*, a library for parallelization of Python workloads. One of Dask's primary use cases is dynamic task scheduling. In Dask, a "task" represents execution of a Python function. In the context of CopScanner, a task is a Python function to interact with a Copernicus trajectory or data file. Dask tasks can be combined to form a task graph, which is a Directed Acyclic Graph (DAG). As CopScanner builds graphs of trajectory scans, it was natural to use Dask to execute these graphs. This specific use case lead to the selection of Dask as the parallelization backend of CopScanner. As will be outlined in this paper, CopScanner modified the task graph creation to happen on-the-fly, rather than pre-scan, to optimize performance. Therefore, the Dask task graph construction was limited in CopScanner, but still used to an extent.

An additional reason for the use of Dask is the abstraction of job scheduling and submission, which is platform agnostic. Job schedulers and submission can often be platform dependent. For example, executing simultaneous processes on a local Windows machine and a high performance computing cluster are vastly different. Dask has several pre-built scheduler objects that can handle various types of job scheduling† (e.g., local, SLURM, PBS) and allow for custom configuration (e.g., number of parallel threads or processes). CopScanner subsequently abstracts the process even more, such that the user simply has to provide a keyword in a configuration file to enable parallelization. The Dask scheduler has remote communication with the tasks, which are Python processes. This enables results from the trajectory scan to be serialized and sent via the network to a scheduler running on a shared network. All results then reside in a central location, which prevents a need for local storage that persists throughout the process of the trajectory scan.

Finally, Dask provides an interactive dashboard during the exeuction of the trajectory scan. Users can see the progress of scan as graph nodes (or tasks, as defined above) are executed. The dashboard works for local trajectory scans or scans on a remote higher performance computer, which is viewed locally via a Secure Shell Protocol (SSH) tunnel.

FILESTORE

Trajectory scans run at a large scale introduce a problem of large data storage and access. Scans can reach millions of trajectories, which creates a necessity to compress and store all data. In addition, querying results becomes difficult.

A key goal of CopScanner was to introduce a paradigm to locally store and access all of the files and metadata generated during a trajectory scan. This was accomplished via the FileStore, which is comprised of two files but interacted with via a single Python object:

- **File Database**: An SQLite‡ database that stores every file produced during a trajectory scan, serialized and compressed
- **MetaData**: A binary Message Pack§ file that stores JSON-serializable values such as trajectory parameters

A large limitation on the FileStore is the need to have local files that can be transferred between analysts. Often trajectory analysis is run at a small scale, where a large, server-based data storage

*https://www.dask.org/
†https://jobqueue.dask.org/en/latest
‡https://www.sqlite.org/
§https://www.msgpack.org/

platform is not needed. The ability for analysts to exchange and store small local files is invaluable flexibility for the individual analyst. This drove the decision to use a SQLite database and a Message Pack key-value store.

As all nodes in CopScanner's scan graph are assigned a Unique Identifier (UID) string (See End-to-End Process section for specifics), that UID is used as a key in both the SQLite database and the metadata msgpack. The user-provided Python code that CopScanner executes to interact with Copernicus trajectories can return files and metadata. All return files are compressed using zlib* and then encoded to Base64, both via Python's standard library. The compressed and encoded files are communicated back to the central scheduler. The metadata is returned in a Python dictionary from the user's provided routine and compressed automatically with Dask to be sent to the scheduler.

The scheduler collects returned files and metadata from executed remote processes. After the scheduler collects a user-defined amount of results, CopScanner writes the results to the FileStore with batched writes. All files are stored as compressed strings in the SQLite database. Each database row has an integer incrementing primary key, the UID of the CopScanner node as an index, the file name, and the compressed file stored as a binary large object (BLOB). As the schema of the returned metadata is not known to CopScanner, any solution to store all files in a local, relational database would be too rigid for this application.

Users interact with the FileStore by performing queries. They can query the FileStore searching for particular metadata fields and results. For example, the FileStore may be queried for all node UIDs that resulted in a Copernicus objective function value less than 0.5. The FileStore provides routines to decode and decompress files which are then written to the local filesystem.

Forward work includes divorcing the FileStore from CopScanner and making it a general Copernicus ideck storage solution, such that Copernicus trajectories and data could be added and removed from a FileStore via Copernicus itself (e.g., from within the GUI). This would enable standalone trajectories to be extracted from the FileStore, loaded in the GUI, edited, and saved once again in the FileStore. With the Copernicus API, this would not require any file system writing of individual Copernicus files. Additionally, the authors are exploring alternate solutions to a single file storage format instead of carrying a SQLite database and a key-value store.

ARCHITECTURE

Scan Structure

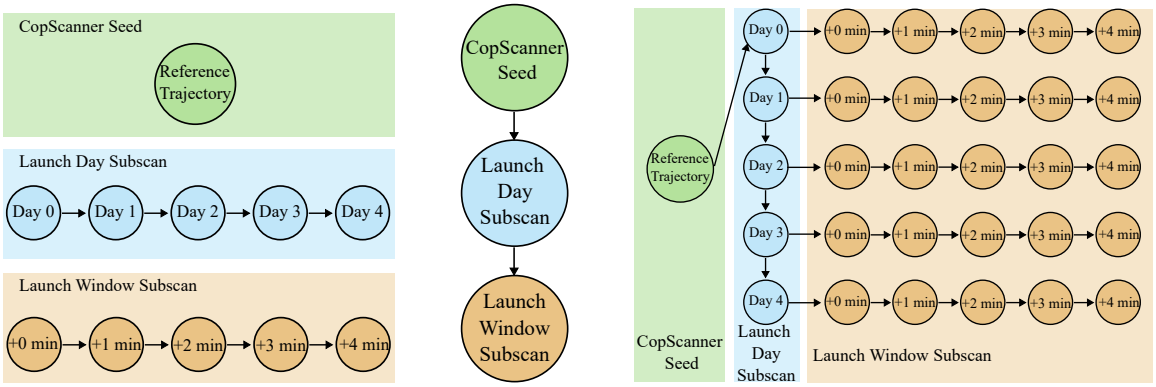
A CopScanner scan is spawned from a set of reference files, called the CopScanner seed. The scan itself is subdivided into a series of different individual subscans. A subscan is constructed from one or more scan parameters, usually corresponding to a parameter within the Copernicus input deck, each of which has a set of values. From these different sets of scan parameters, a set of subscan nodes is generated where each node is defined from a unique combination of the different scan parameter values (i.e., the set of nodes is equivalent to the Cartesian product over each of that subscan's scan parameter value sets). From these nodes, the directed subscan graph is constructed. The edges of this graph are defined by the subscan dependencies, for which CopScanner offers built-in support for simple increasing/decreasing relationships between scan parameter values. However, users may also specify their own dependency with a custom Python module, allowing for arbitrary specification of the subscan dependency structure.

*https://www.zlib.net/

The individual subscans form the overall CopScanner scan via the scan chain. The scan chain is a directed graph with the subscans as its nodes. Each subscan has exactly one scan chain dependency on another subscan or the CopScanner seed. This is referred to as that subscan's source. Because of this specification, the scan chain is a special type of directed graph, called an arborescence or rooted polytree with the CopScanner seed as its root node.

The overall scan graph, or scan DAG, is generated by a so-called "tesselation" of the individual subscan graphs onto the scan chain. This tesselation occurs by each subscan in the scan chain spawning a copy of its subscan graph off of a certain subset of the nodes in its source subscan. This subset shall be called the source node set. The source node set can either be every node in the source subscan (called "default spawning") or it can be only the end nodes, i.e., the nodes in the source subscan which have no successors in the source subscan (called "end spawning")*. This process generates a recursive pattern of subscan graphs as subscans further down the scan chain add successivley more branches to the scan DAG. For the purposes of the tesselation, the CopScanner seed is treated as a subscan with a single scan parameter, being the input file, and a subscan graph consisting of disconnected nodes containing each of the seed files.

It is useful to illustrate this proccess with a straightforward, conceptual example. This example is a simple launch epoch mission scan with a single reference trajectory as the CopScanner seed. For this scan there are two subscans: a launch day subscan and a launch window subscan. The launch day subscan has a single scan parameter being the mission launch epoch, for which there are values at one day increments over a total of five days. The launch window subscan also has the mission launch epoch as its single scan parameter with one minute increments over a total span of five minutes. Both subscans have a total of five subscan nodes, use the built-in increasing subscan dependency in CopScanner (i.e., each launch epoch scan parameter value is dependent on the value that directly precedes it, chronologically). Finally, the launch window subscan is dependent on the launch day subscan (equivalently, the launch window subscan's source is the launch day subscan), the launch day subscan is dependent on the CopScanner seed, and all subscans use the default spawning. Figure 1 shows the corresponding subscan graphs, scan chain, and scan DAG for this example.



(a) Subscan Graphs. **(b) Scan Chain.** **(c) Scan DAG.**

Figure 1: Launch Epoch Scan Example.

*For example, if the launch window subscan in Figure1 were specified in this way, then only the launch day subscan node "Day 4" would have spawned a launch window subscan.

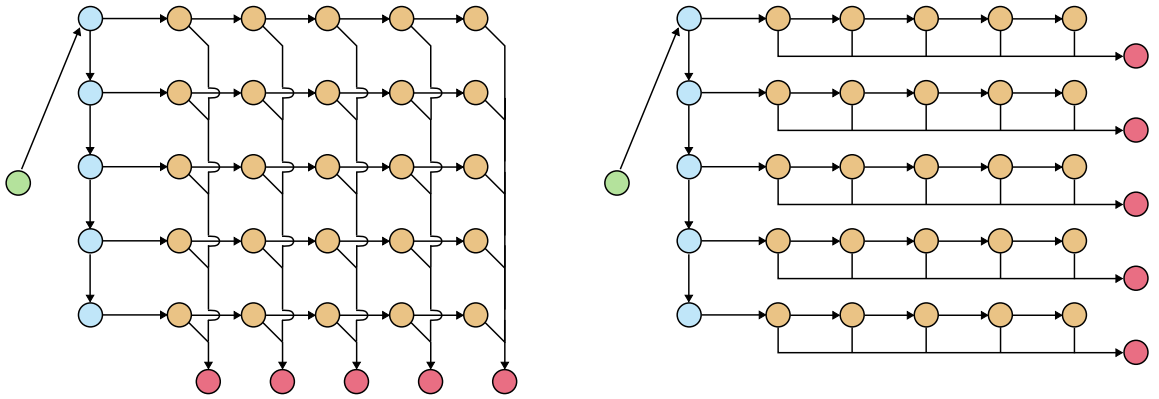
From this example, the scan graph tesselation process is as follows: each seed file (in this case, the single reference trajectory), spawns a copy of the launch day subscan graph where the mission launch epoch is advanced by the corresponding scan parameter value. Then, each node in the launch day subscan spawns a copy of the launch window subscan graph where the mission launch epoch is again advanced by the corresponding scan parameter value (this time by minute increments rather than days). The scan DAG for this example is shown in Figure 1c.

It is important to note the difference between the nodes in the scan DAG and the nodes in the subscan graphs. We refer to the bottom right node in Figure 1c ("Day 4 +4 min") as being a member of the launch window subscan in the scan graph, or scan DAG, and this node is distinct from the "+4 min" node in the launch window subscan graph in Figure 1a. The former case represents a realization of the latter where an actual process has been executed with the corresponding scan parameter value input and associated metadata and files have been generated. In effect, the launch window subscan graph nodes represent a plan while the launch window subscan nodes in the scan graph are the fulfillment of that plan.

Subscans can be one of three types:

- **"One to n"** scans will spawn a new version of their subscan graph off of each node of its source node set.
- **"Open ended"** scans are similar to "one to n" scans except that one of their scan parameters is an open value that will be indefinitely incremented during execution to generate new subscan graph nodes until the callable returns a kill flag in the metadata or until a certain bound value is reached. This type of scan is useful, for example, for a schema where the desired range of scan parameter values may be dependent on the values of optimization variables from a previous subscan.
- **"M to n"** subscans are convergence points in the scan DAG (for this reason the phrase "convergent subscan" will be used interchangeably with "m to n subscan"). They spawn single copies of their subscan graph off of subsets of their source node set. These subsets are members of a partition of the source node set which is generated from an equivalence relation on the source node set. Each "m to n" subscan has a set of dividing scan parameters which are used to define this equivalence relation. Each of the dividing scan parameters must be one of the scan parameters from a subscan among the ancestors of the "m to n" subscan in the scan chain. The subscans in the scan chain which contain the dividing scan parameters are called dividing subscans. For two nodes i and j in the source node set, then $i \sim j$ if the values of all the dividing scan parameters in the relevant scan parameter histories of i and j are equal. Figure 2 shows two different configurations for how a convergent subscan might be added to the launch epoch scan example.

Because there are no constraints on the structure of the subscan graph dependencies, a subscan graph may not be a DAG, in which case it may not have a well-defined 'starting point' from which to spawn a new subscan layer. However any directed graph can be condensed into a DAG by contracting each of the strongly connected components of the initial directed graph into a single node (see Figure 3). Therefore, a set of starting nodes can be determined for each scan by collecting all of the nodes which are members of strongly connected components which have no in-degrees from other strongly connected components. In addition, for scan parameters which are recognized as Copernicus input deck variables, CopScanner will find the node from each strongly connected component with values closest to the values in the source node's input deck and use that node to



(a) Here the dividing scan parameter of the convergent subscan is the launch window subscan launch epoch. The equivalence classes form "columns" in the scan DAG, and identical launch minutes across different launch days are aggregated in each of the convergent subscan nodes.

(b) Here the dividing scan parameter of the convergent subscan is the launch day subscan launch epoch. The equivalence classes form "rows" in the scan DAG, and all of the launch minutes from the same launch day are aggregated in each of the convergent subscan nodes.

Figure 2: Launch Epoch Scan example with convergent subscan. This figure shows the scan DAG for the launch day scan example from Figure 1, with an additional convergent subscan (shown in red) after the launch window subscan with a single node in its subscan graph.

spawn a new scan level. Users may wish to specify a subscan graph in this way if one or more of the subscan scan parameters is a recognized Copernicus input deck variable and its value in the initial guess trajectory may be variable due to a previous subscan process.

Finally, during the overall run, each edge of the subscan graph can only be traversed once, and each node can only be run once. This ensures that the overall scan graph is acyclic even if some or all of the individual subscan graphs are not. The use of cyclical subscan graphs allows users to generate versatile schema for optimizing complex trajectory scans that prioritizes jobs that converge more quickly as initial guesses for successive nodes.

End-to-End Process

An overview of the CopScanner architecture is shown in Figure 4. As discussed, a scan in CopScanner is executed as a DAG of related tasks. Nodes in the DAG correspond to jobs, and the directed edges denote dependencies between these jobs. Each node within the CopScanner DAG is executed on a Dask worker. For a typical use case, each node will receive as inputs from the scheduler a set of dependency files (typically a Copernicus trajectory file with associated plugin files), and a JavaScript Object Notation (JSON) metadata structure with that node's inputs and dependencies. All trajectory file manipulation and optimization is handled on the Dask worker by a separate Python module referred to as the *callable*. CopScanner provides a default callable which can handle many typical Copernicus input deck operations, but users may also write and specify their own callables for each subscan. The support for custom callables is what makes CopScanner a truly versatile trajectory optimization scan software. In particular, while this document focuses on how CopScanner was built for parallelization of Copernicus trajectory scans, there is no requirement that anything in the callable be Copernicus-related. Therefore, practically any task that can be

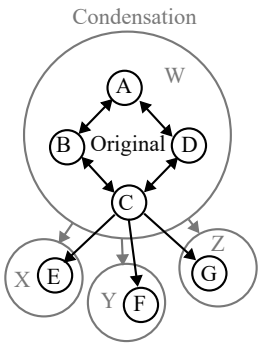


Figure 3: Example Subscan graph condensation for determining starting node. Given the initial subscan graph (black), CopScanner will generate the condensation (grey). Each node in the condensation corresponds to a set of nodes in the original graph: $W : \{A, B, C, D\}$, $X : \{E\}$, $Y : \{F\}$, $Z : \{G\}$. The condensed graph is a DAG by definition for which a set of starting nodes ($\{W\}$) can be unambiguously identified. All of the corresponding nodes in the original graph are chosen as the starting nodes for this subscan, unless all of the scan parameters are recognized Copernicus input deck variables, in which case, for every copy of this subscan graph that is spawned CopScanner will chose the node in the set $\{A, B, C, D\}$ with scan parameter values closest to those of the initial guess input deck.

framed as a DAG of independent processes can be run with CopScanner.

CopScanner runs are generated from a set of four user-provided JSON configuration files:

- **The seed configuration** defines the input to the scan, usually a single Copernicus input deck, or a FileStore from another CopScanner run. The latter case represents a CopScanner functionality, known as bootstrapping, that allows incomplete runs to be continued from where they left off or complete runs to be extended, as described in the Bootstrapping Section.
- **The output configuration** is a simple file that defines the location and name of the scan FileStore and other scheduler outputs.
- **The hardware configuration** is predominantly a pass-through for a Dask cluster object.
- **The scan configurations** define the different subscans that compose the overall CopScanner scan (see Scan Structure section). Each of the individual subscans are defined by their own JSON input, which defines their scan parameters, dependencies, and callable script.

During execution, new jobs are sent to the Dask scheduler, which distributes them to the Dask cluster for execution. Upon completion, the jobs are returned to the CopScanner scheduler via the Dask scheduler. Each job in CopScanner can be broken down into three components: a set of scan parameters and corresponding values, a set of file dependencies (e.g., the Copernicus input deck and its plugin files), and a record of that job's ancestors' scan parameter values and their corresponding order in the scan graph. The latter is referred to as the scan parameter history of a job. It is necessary for each job to keep track of its own scan parameter history so that it can execute independently of the scheduler with complete knowledge of its own relationship to the root node. In addition, the scan parameter history allows us to index each job in both the overall scan DAG as well as the FileStore for efficient ($O(1)$) lookup by generating a unique UID string for each job using the subscan name, scan parameter key, and corresponding scan parameter value from a specific subset of the scan parameter history, called the *relevant* scan parameter history. The relevant scan parameter history

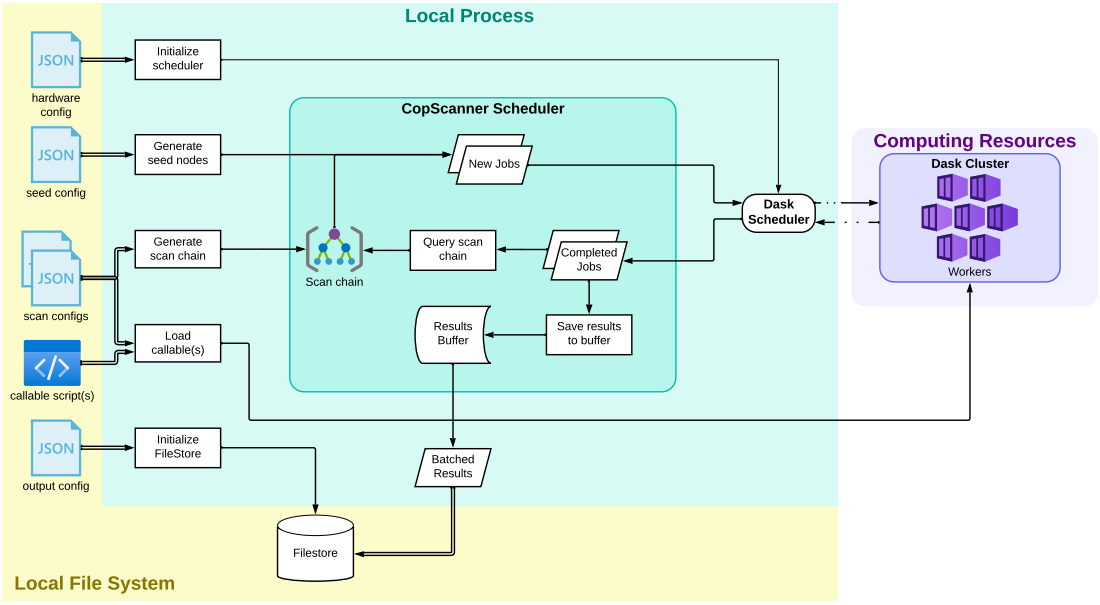


Figure 4: Overview of a CopScanner run.

is the scan parameter values from each of a job's most recent ancestors, including itself, at each subscan level preceding it. However, if at any previous subscan level, a node has more than one most recent ancestor* with different values for a particular scan parameter, then that scan parameter is not included in the relevant scan parameter history. This is because the existence of multiple different values for a scan parameter among a node's most recent ancestors at a particular subscan level indicates that there must be a convergence step somewhere between those ancestors and the current node. In this case, each of those duplicate ancestors can be uniquely identified by that convergence step, which will itself be another most recent ancestor of the current node. Another way to conceptualize this process is to think of the UID as a coordinate in a discrete, n-dimensional space where each dimension is one of the scan parameters in the relevant scan parameter history, and the discretization of each dimension is the set of possible values for that scan parameter. Figure 5 provides a graphical example of these concepts.

When the CopScanner scheduler receives a completed job, it stores the results to a buffer in RAM where it will be held until the next chunked write to the FileStore. The scheduler will also query the scan chain to generate any dependent new jobs. The process of querying the scan chain is explained in the following sections.

The FileStore is the main output of a CopScanner run, and can also be used as an input to CopScanner, in order to restart a scan from a previous benchmark. This functionality is referred to as "bootstrapping", and represents an integral utility in the software architecture. In particular, the desire to be able to restart incomplete or otherwise interrupted scans from any arbitrary benchmark

*Here it should be made clear that *a node's most recent ancestor at a previous subscan level* is meant as any node at that previous subscan level for which there is a path through the DAG from that ancestor to the node in question which does not pass through any other nodes at that previous subscan level. This definition does not require that the path lengths from each of a node's most recent ancestors at a particular subscan level be equal.

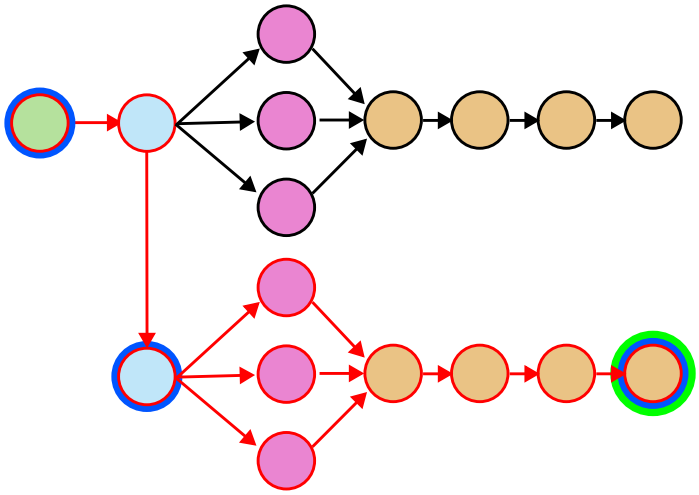


Figure 5: Example of the scan parameter history. This figure shows an example scan DAG with three arbitrary subscans (the green-filled node is the CopScanner seed), each with a single scan parameter. The orange subscan is a convergent subscan with the red subscan's scan parameter as its only dividing scan parameter. The scan parameter history (red borders) and relevant scan paramter history (blue borders) are shown for the bottom right node (green border). Note that the node in question is a member of its own relevant scan parameter history and that none of the nodes from the red subscan are included in the relevant scan parameter history, because the different scan parameter values where converged by the orange subscan.

impacted the algorithmic design of much of the CopScanner scheduler's internal logic.

All graphs and their associated operations are implemented in CopScanner using the NetworkX* Python library.

The Scheduler Process

The main execution loop of the CopScanner scheduler process involves receiving completed jobs from the Dask scheduler, initializing dependent new jobs, and then submitting those to the Dask scheduler. The middle step in this process, dubbed "querying the scan chain", is where the bulk of the scheduler's logic resides and is the means by which the scan chain "tesselation" process actually occurs.

Querying the scan chain can be split into two steps. First, CopScanner queries the subscan graph of the same subscan as the completed job for any dependent nodes, and then it queries all of the subscan that are dependent on the completed job's subscan for their starting nodes if the completed job is a part of that subscan's source node set. The results of both of these queries are a set of nodes (each with an associated subscan), set of scan parameters and corresponding values, and default metadata. These nodes are then given a scan parameter history, using the completed job's scan parameter history and dependencies using the completed job's outputs to generate a set of new jobs for execution. Before submitting these new jobs, CopScanner first checks that there are no duplicate jobs already submitted or completed in the scan. It accomplishes this via the UID strings which, as previously described, are formulated such that two jobs with the same relevant scan parameter

*https://networkx.org/

histories would have the same UID without necessarily having the same dependencies. As will be discussed in the proceeding section, the overall DAG may not be a continuous graph. Particularly, it is possible that during a bootstrapped run, one or more of these new jobs has already been executed, in which case an iterative Breadth-First Search (BFS) is performed through the DAG starting at these jobs to find unexecuted new jobs. Although, care needs to be taken not to generate any cycles or run any jobs twice.

There is a further complication in the second step of querying the scan chain if any of the dependent subscans are an "m to n" or convergent subscan. Because the starting nodes of these subscans will generally have multiple predecessors in the scan DAG, the scheduler needs to be sure that no further jobs in the completed job's equivalence class, defined by the convergent subscan's dividing scan parameters, could be generated. Furthermore, due to bootstrapping, it cannot be assumed that the scan graph is contiguous, so an algorithm must be developed that is robust to a potentially disconnected DAG

Given a completed job from the source subscan of a convergent subscan, CopScanner needs to determine if it is the last completed job in its equivalence class defined by the values of the dividing scan parameters of the convergent subscan in question. First, define the dividing scan parameter history as the subset of the completed job's relevant scan parameter history that includes only the convergent subscan's dividing scan parameters. Next, define the dividing node set as every node in the scan DAG that is a member of one of the convergent subscan's dividing subscans and whose relevant scan parameter history matches every scan parameter in the dividing scan parameter history of the completed job* (See Figure 6 for example). The dividing node set makes up the most recent ancestors in the dividing subscans of all of the nodes in the completed job's equivalence class. There is no guarantee that all of the dividing nodes have been executed or even generated, so the algorithm will begin by performing a modified BFS from the roots (nodes with in-degrees = 0) of the scan DAG to find all dividing nodes or all of their incomplete ancestors. This modified BFS starts by filtering the root nodes to only those which could possibly have the dividing nodes among their ancestors in the scan DAG. While generally it is not possible to guarantee that a root node will have a dividing node among its ancestors, the dividing scan parameter history dictates which nodes are guaranteed to not have dividing nodes among their ancestors. Specifically any root node which has a relevant scan parameter history that does not have matching values at every scan parameter to the dividing scan parameter history, except for perhaps the scan parameters of that root node's subscan, will not have any dividing nodes among its ancestors. Note that scan parameters of each root node's subscan need not necessarily match the dividing scan parameter history because they may have descendants at the same subscan level which do match.

*For example, the dividing node set of the green-bordered node in Figure 5 is all of the nodes from the red subscan with red borders

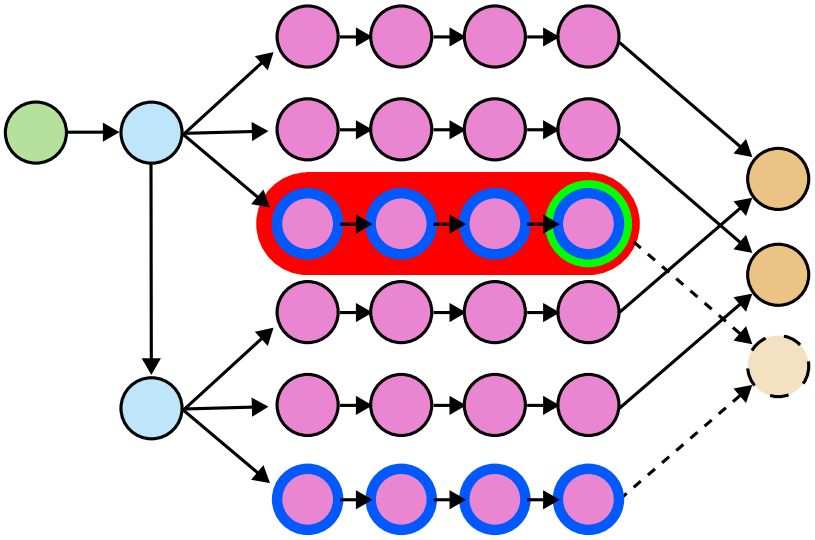


Figure 6: Convergent subscan spawning example. This figure shows an example scan DAG with three arbitrary subscans (the green-filled node is the CopScanner seed). The blue subscan has a single scan parameter with two values. The red subscan has two scan parameters with three and four values each, represented as the number of rows and columns in each copy of its subscan graph, and the orange subscan is a convergent subscan with end spawning, and the first scan parameter of the red subscan (i.e., the one with only three different values) as its only dividing scan parameter. For the given completed node (green border), the dividing nodes (blue borders) and dividing scan parameter history (red border) are shown. The new convergent subscan job that will be run is shown with dashed borders.

Once the root nodes have been properly filtered, they are used to form a queue to start the modified BFS. The outputs of the iteration are two sets: a set of incomplete nodes which could contain dividing nodes among their ancestors and a set of dividing nodes. At each step of the BFS iteration, four separate conditions are checked:

(i) if the current iterate's subscan is one of the dividing subscans
(ii) if the current iterate's subscan is one of the ancestors of a dividing subscan in the scan chain
(iii) if the current iterate's scan parameters' values match those in the dividing scan parameter history
(iv) if the job associated with the current iterate is complete

If both (i) and (ii) are false, then the current iterate's subscan is neither a dividing subscan nor could it have any dividing nodes among its ancestors, so the BFS stops expanding at that node. If (iii) is true and (iv) is false, then the current iterate is added to the set of incomplete nodes. Finally, if both (i) and (iii) are true, then the current iterate is a dividing node and it is added to the set of dividing nodes.

After the iteration halts, if any of the incomplete nodes are from one of the dividing subscans or if no dividing nodes were discovered, then the "m to n" subscan is not ready to spawn another subscan graph off of the current completed job. However, if the iteration returned non-empty sets of both dividing and incomplete nodes without any overlap, then CopScanner needs to check whether there are any missing dividing nodes. This is done by iterating over each of the dividing subscans in their

topological order in the scan chain. For each of these dividing subscans, if any of the incomplete nodes are from that subscan or any subscan preceding it in the scan chain, then CopScanner checks if any dividing nodes at this dividing subscan level are missing. To do this, it generates all possible combinations of every value of all scan parameters which are not dividing scan parameters and which belong to the given dividing subscan or a subscan that preceeds it in the scan chain. Then each of these combinations is checked against the scan parameter values of the relevant scan parameter histories of all of the dividing subscan's completed dividing nodes', collected during the modified BFS. If any of the expected combinations are missing from the set of dividing nodes generated during the modified BFS, then the "m to n" subscan level is not ready to start.

If the modified BFS iteration only returned a non-empty set of dividing nodes and there were no incomplete nodes found, then there is no need to perform the same check of the scan parameter values, because there is not a possibility that more dividing nodes will be generated for the equivalence class in question.

The final step of the algorithm is to perform a second BFS from the dividing nodes to the nodes of the source subscan, terminating if any incomplete nodes are encountered.If this iteration finishes without premature termination, then there are no more jobs to be run which could be a part of the completed job's equivalence class, and a new "m to n" subscan level can be spawned.

When the scheduler determines that an "m to n" subscan is not ready to begin based on a given completed job, it needs to store the results of that job so that when it is ready to begin it can quickly access it at a later time to pass out the the starting node(s) of the new subscan layer. Much like the solution to indexing the individual nodes, CopScanner uses the dividing scan parameter history to generate UID strings for indexing the partitions of the source nodes for constant look up time.

Bootstrapping

"Bootstrapping" refers to the process of starting a new scan using the results from a previously run scan. The most common use cases for bootstrapping are to continue a run that only partially completed or to extend the results of a completed scan by either extending the ranges of values of one or more of the scan parameters or by adding additional subscans to the scan chain. Fundamentally, however, bootstrapping is the process by which a CopScanner FileStore is used as the seed of a new scan. In concept, the process is simple enough: from a seed FileStore, generate a bootstrapped scan DAG and then continue the execution of that DAG. The only problem with this formulation is determining the bootstrapped seed nodes, which are the nodes in the bootstrapped DAG that are active (i.e., which nodes have unexecuted successors). The only way to do this is to query the scan chain with each completed node in the seed FileStore. However, if this process is implemented naively, it is possible that multiple of the new jobs generated in this step will be duplicates of already completed jobs. In addition, for each completed job from the seed FileStore a scan parameter history must be pre-generated, and the only way to faithfully generate a scan parameter history for one of these seed nodes is to traverse all paths in the seed FileStore's DAG from the root node to that node. For highly connected and/or deep DAGs this process can become exceedingly inefficient.

The solution to these two problems is to group the seed nodes into connected components, and then only query the scan chain with the roots (in-degrees = 0) of those connected components. These are the true bootstrap seeds. A new processing step is added for each new job generated during the query step, where the scheduler checks if the new job is a duplicate of any already completed/generated jobs.If the new job is a duplicate of a completed job, the scan chain is queried

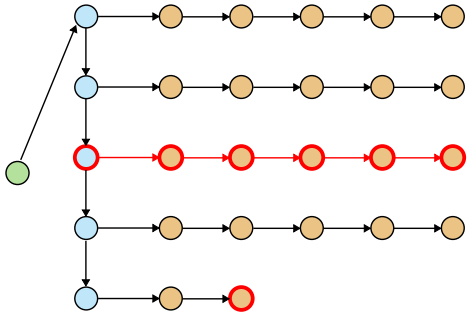
again with that job to look for more new jobs. Thus the querying process becomes a BFS through the scan DAG. At each step in the BFS, CopScanner checks whether or not the duplicate job has any common ancestors with the original completed jobs. This ensures that the overall scan graph remains acyclic. Consequently, we must then remove all of the edges from the bootstrapped DAG before querying the scan chain with our bootstrapped seeds so that the BFS correctly traverses the completed nodes in the seed FileStore. In addition this affords users the capability to modify the subscan graph dependencies of one or more of the subscans in order to try different initial guesses for the callable continuation method for unconverged nodes from the original run.

CopScanner offers three different configuration options for adjusting the specificity of the bootstrapped run: the starting subscan, the seed filters, and the scan filters. The starting subscan is the subscan from which the bootstrap run will start, and only that subscan and those downstream of it in the scan chain will be involved during the bootstrapped run. The seed filters and scan filters are a set of custom FileStore-specific JSON syntax metadata filter conditions. Any nodes in the seed FileStore which do not pass the seed filters will be removed and re-run while any nodes that do not pass the scan filters will terminate the recursive BFS during the scan chain querying step, thus halting the scan at that node. These options afford users the ability not just to restart scans from their last saved point, but also to refine and more precisely target their restarts to impact specific nodes in the scan.

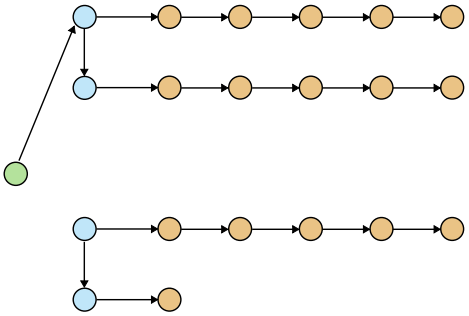
The bootstrapping process is as follows. Given a seed FileStore, CopScanner generates the DAG, G, from that FileStore. Then define S_0 as the set of all nodes in G that pass the seed filters (if no seed filters then S_0 is all of the nodes in G), and G_0 as the subgraph of G formed from S_0. Next define the set S_1 as all nodes in S_0 which are from the starting subscan or one of its descendants in the scan chain and which pass the scan filters if any are specified. The bootstrap seed nodes S_b are all of the nodes in S_1 which have no predecessors in S_1. Finally, define the bootstrapped DAG G_b as a copy of G_0 with all edges removed that start or end at any node in the set $S_1 \setminus S_b$. Once this process has been performed, the scan chain can be queiried with each node in S_b, using G_1 as the scan DAG. Figure 7 shows an example scan graph following this algorithm.

CONCLUSIONS AND FUTURE WORK

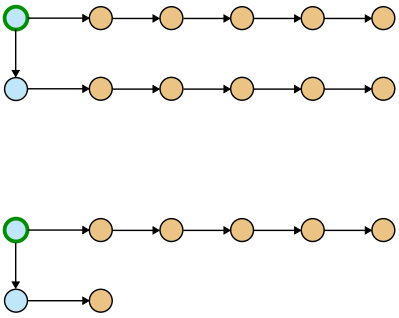
CopScanner, a new component of the Copernicus trajectory optimization system, has been described. Early versions have been used at JSC for trajectory scans for future Artemis missions, and it will be included in a future Copernicus release to all users. The ability to easily set up and execute massive trajectory optimization scans is a critical feature that, heretofore, individual users have had to devise solutions for on an ad hoc basis. It is hoped that including this capability in Copernicus itself will be an important enabling technology for the next generation of advanced mission design tasks. As the main orthodoxy of ongoing CopScanner development is to emphasize its adaptability, future work will focus on further abstraction and modularization of the scan architecture as presented. This includes increasing the versatility of CopScanner with features such as allowing users to specify custom schema for the source node sets of their subscans, or enabling convergent subscans to have multiple dependencies in the scan chain.



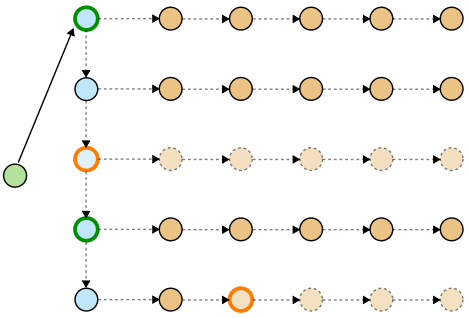
(a) A possible execution of the Launch Epoch Scan from Figure 1 where the "Day 4 + 1 min" node (red border, bottom right) encountered an error during execution and the subsequent nodes in that thread were not executed as a result, and the "Day 2" solution thread (red border, middle row) optimized to an undesirable solution.



(b) The graph G_0. In order to re-run the erroneous nodes, the seed filters are configured to filter out the red-bordered nodes from 7a



(c) For this example, there are no scan filters, and the launch day subscan is chosen as the starting subscan[a]. This figure depicts the subgraph of G_0 formed from S_1. The bootstrap seed nodes (S_b) are shown with green borders.



(d) The bootstrapped DAG G_b that will be used to query the scan chain. The edges that were removed for starting or ending in the set $S_1 \setminus S_b$ are shown with dotted gray lines, the nodes that are not yet executed are shown with low saturation, the bootstrap seeds (S_b) have green borders, and the new jobs that will be generated after querying the scan chain are shown with orange borders.

[a]Note that the CopScanner seed could have also been chosen as the starting subscan without meaningfully impacted the result of this bootstrap

Figure 7: Bootstrap of the Launch Epoch Scan Example.

ACKNOWLEDGMENT

This work was partially funded by NASA JSC under contract NNJ13HA01C. The authors wish to thank Sarah Smallwood, Chris Foster, and Colin Brown for their assistance with this work.

TERMINOLOGY

BFS	Breadth-First Search
BLOB	binary large object
DAG	Directed Acyclic Graph
EI	Entry Interface
GUI	Graphical User Interface
JSC	Johnson Space Center
JSON	JavaScript Object Notation
NASA	National Aeronautics and Space Administration
PBS	Portable Batch System
PPE	Power and Propulsion Element
RAM	Random-Access Memory
SLURM	Simple Linux Utility for Resource Management
SSH	Secure Shell Protocol
TIG	Time of Ignition
TLI	Trans-Lunar Injection
UID	Unique Identifier

REFERENCES

[1] J. Williams, A. Kamath, R. Eckman, G. Condon, R. Mathur, and D. Davis, "Copernicus 5.0: Latest Advances in JSC's Spacecraft Trajectory Optimization and Design System," AAS/AIAA Astrodynamics Specialist Conference, Portland, ME, Aug. 2019. AAS 19-719.

[2] J. P. Gutkowski, T. F. Dawn, and R. M. Jedrey, "Evolution of Orion Mission Design for Exploration Mission 1 and 2," 39th Annual AAS Guidance, Navigation and Control Conference, 2016. AAS 16-111.

[3] T. F. Dawn, J. P. Gutkowski, A. L. Batcha, J. Williams, and S. M. Pedrotty, "Trajectory Design Considerations for Exploration Mission 1," AIAA/AAS Space Flight Mechanics Meeting, Jan. 2017.

[4] A. L. Batcha, J. Williams, T. F. Dawn, J. P. Gutkowski, M. V. Widner, S. L. Smallwood, B. J. Killeen, E. C. Williams, and R. E. Harpold, "Artemis I Trajectory Design and Optimization," AAS/AIAA Astrodynamics Specialist Conference, Aug. 2020. AAS 20-649.

[5] N. L. Parrish, E. Kayser, S. Udupa, J. S. Parker, B. W. Cheetham, and D. C. Davis, "Survey of Ballistic Lunar Transfers to Near Rectilinear Halo Orbits," AAS/AIAA Astrodynamics Specialist Conference, Portland, ME, Aug. 2019. AAS 19-740.

[6] S. L. McCarty and M. McGuire, "Parallel Monotonic Basin Hopping for Low Thrust Trajectory Optimization," *28th AIAA/AAS Space Flight Mechanics Meeting*, Jan. 2018. AIAA 2018-1452.

[7] G. L. Condon, C. A. Ocampo, L. Burke, C. C. Esty, C. Berry, B. Mahajan, and S. P. Downs, "Mission and Trajectory Design Considerations for a Human Lunar Mission Originating from a Near Rectilinear Halo Orbit," AIAA Scitech 2020 Forum, Jan. 2020. AIAA 2020-1921.

[8] R. Harpold, C. Brown, B. J. Killeen, R. A. Eckman, T. F. Dawn, B. Adebonojo, and J. Williams, "Artemis I Off-Nominal Trajectory Design," AAS/AIAA Astrodynamics Specialist Conference, Aug. 2023. AAS 23-129.

[9] J. Arrieta, "Applications of Graphs in Trajectory Design," International Symposium on Space Flight Dynamics, ISSFD, Oct. 2015.